Course Outline

Day1

- Introduction to Java
- Basic Java Concepts

Day2

- Data Types & Operators
- using Arrays & Strings
- Controlling Program Flow
- Modifiers and Access Specifiers

Day3

- Simple GUI
- Essential Java Classes
- Java Exception

Day4

- Interfaces
- Multi-Threading

Day5

- Inner class
- Event Handling

Java Programming
Interfaces

Presented By
Dr. Eman Hesham, DBA.

Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

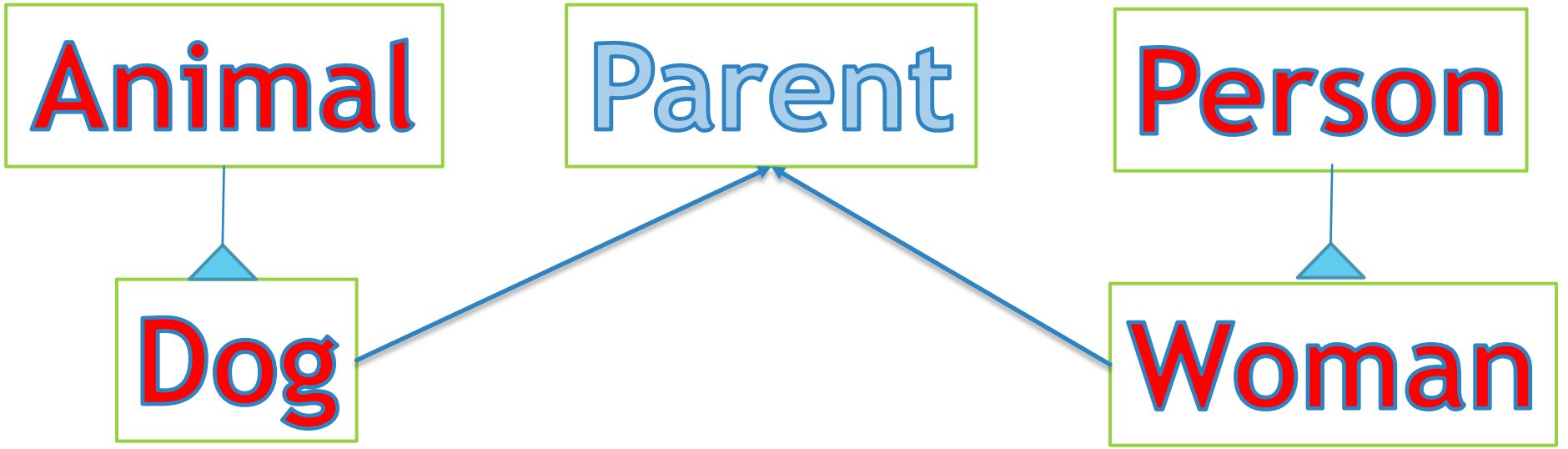
Interfaces

- In OOP, it is sometimes helpful to define what a class must do but not how it will do it.
- An abstract method defines the signature for a method but provides no implementation.
- A subclass must provide its own implementation of each abstract method defined by its superclass.
- Thus, an abstract method specifies the *interface* to the method but not the *implementation*.
- In Java, you can fully separate a class' interface from its implementation by using the keyword **interface**.

Interfaces

- An ***interface*** is syntactically similar to an abstract class, in that you can specify one or more methods that have no body.
- Those methods must be implemented by a class in order for their actions to be defined.
- An *interface* specifies what must be done, but not how to do it.
- Once an interface is defined, any number of classes can implement it.
- Also, one class can implement any number of interfaces.

Interfaces



Interfaces

Here is an example of an **interface** definition.

```
public interface Numbers
{
int getNext(); // return next number in series
void reset(); // restart
void setStart(int x); // set starting value
}
```

Implementing Interfaces

The general form of a class that includes the **implements** clause looks like this:

```
Access Specifier  class classname extends superclass implements interface {
// class-body
}
```

Implementing Interfaces

```
// Implement Numbers.
class ByTwos implements Numbers
{
    int start;
    int val;
    Public ByTwos() {
        start = 0;
        val = 0;
          }
    public int getNext() {
        val += 2;
        return val;
     }
    public void reset() {
        val = start;
    }
    public void setStart(int x) {
        start = x;
        val = x;
    } }
```

- Class **ByTwos** implements the **Numbers** interface
- Notice that the methods getNext( ), reset( ), and setStart( ) are declared using the public access specifier

Implementing Interfaces

```
// Implement Numbers.
class ByThrees implements Numbers
{
    int start;
    int val;
      public ByThrees() {
        start = 0;
        val = 0;
      }
    public int getNext() {
        val += 3; return val;
    }
    public void reset() {
    val = start;
    }
    public void setStart (int x) {
    start = x;
     val = x;
    }
}
```

- Class **ByThrees** provides another implementation of the **Numbers** interface
- Notice that the methods getNext( ), reset( ), and setStart( ) are declared using the public access specifier

Using interface reference

```
class Demo2
{
   public static void main (String args[])
   {
        ByTwos twoOb = new ByTwos();
        ByThrees threeOb = new ByThrees();
        Numbers ob;
        for(int i=0; i < 5; i++) {
                ob = twoOb;
                System.out.println("Next ByTwos value is " +
ob.getNext());
                ob = threeOb;
                System.out.println("Next ByThrees value is " +
ob.getNext());
        }
   } }
```

General Consideration about interfaces

- Variables can be declared in an interface, but they are implicitly public, static, and final.
- To define a set of shared constants, create an interface that contains only these constants, without any methods.
- One interface can inherit another by use of the keyword extends. *The syntax is the same as for inheriting classes.*
- When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Default Methods

- Prior to JDK 8, an interface could not define any implementation whatsoever.
- The release of JDK 8 changed this by adding a new capability to interface called the default method.
- A default method lets you define a default implementation for an interface method.
- You specify an interface default method by using the default keyword

Default Methods

```
interface InterfaceA {

    public void saySomething();

    default public void sayHi() {
      System.out.println("Hi");
    }

}
```

```
public class MyClass implements InterfaceA
{

    public void saySomething() {
        System.out.println("Hello World");
    }

}
```

Default Methods

Extending Interfaces That Contain Default Methods

- Not mention the default method at all, which lets your extended interface inherit the default method.

```
interface Intf1 {
    default void method() { doSomething();  }
}

interface Intf2 extends Intf1 {

}
```

Default Methods

- **Extending Interfaces That Contain Default Methods**
  - Redefine the default method, which overrides it.

```
interface Intf1 {
    default void method() { doSomething();  }
}

interface Intf2 extends Intf1 {
    default void method() { doAnother();  }
}
```

Default Methods

Extending Interfaces That Contain Default Methods

- Re-declare the default method, which makes it abstract.

```
interface Intf1 {
    default void method() { doSomething();  }
}

interface Intf2 extends Intf1 {
    abstract void method();
}
```

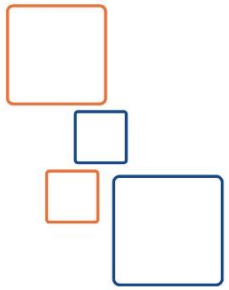
Use static Methods in an Interface

- JDK 8 added another new capability to interface:
  - the ability to define one or more static methods.
- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- Thus, no implementation of the interface is necessary, and no instance of the interface is required in order to call a static method.

Use static Methods in an Interface

```
public interface MyIF {
// This is a "normal" interface method declaration.
//It does NOT define a default implementation.
    int x=10;
    int getUserID();

// This is a default method. Notice that it provides a
//default implementation.
    default int getAdminID()
    {
        return 1;
    }
// This is a static interface method.
    static int getUniversalID()
    {
        return 0;
    }
}
```

Java Programming
Multi-Threading



Java™ Education
and Technology Services

Invest In Yourself ,
Develop Your Career

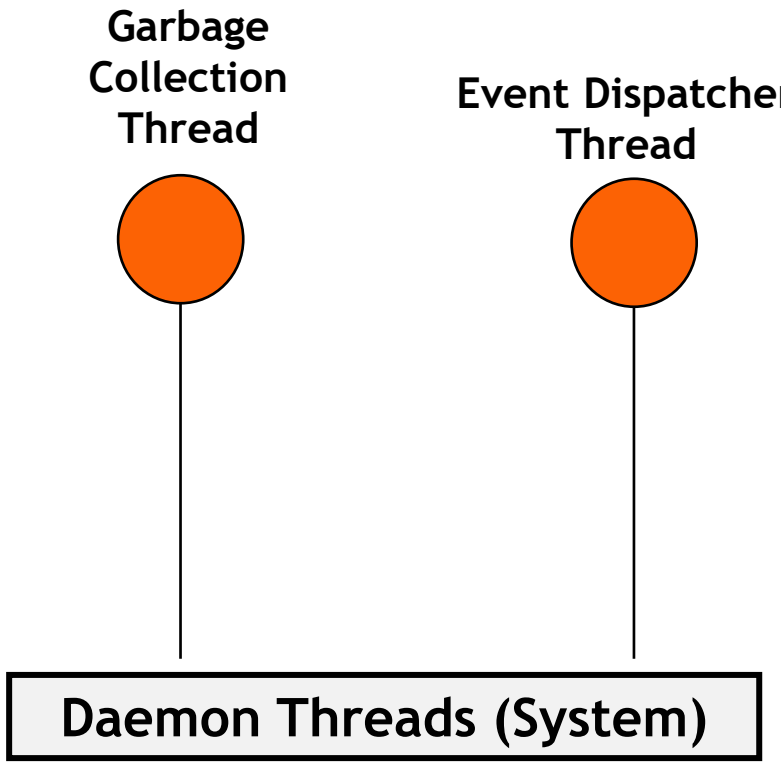
What is Thread?

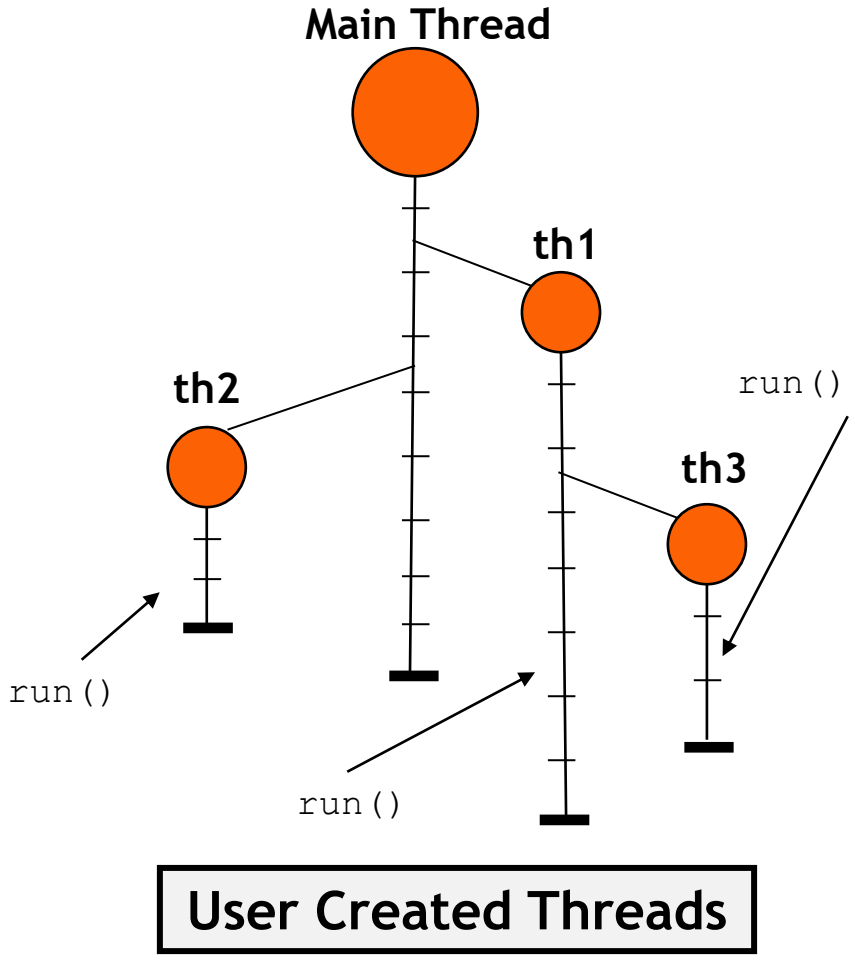
- A Thread is
  - A single sequential execution path in a program
  - Used when we need to execute two or more program segments concurrently (multitasking).
  - Used in many applications:
    - Games, animation, perform I/O
  - Every program has at least two threads.
  - Each thread has its own stack, priority & virtual set of registers.

- Multiple threads does not mean that they execute in parallel when you're working in a single CPU.
  - Some kind of scheduling algorithm is used to manage the threads (e.g. Round Robin).
  - The scheduling algorithm is JVM specific (i.e. depending on the scheduling algorithm of the underlying operating system)

Threads

- several thread objects that are executing concurrently:





Threads

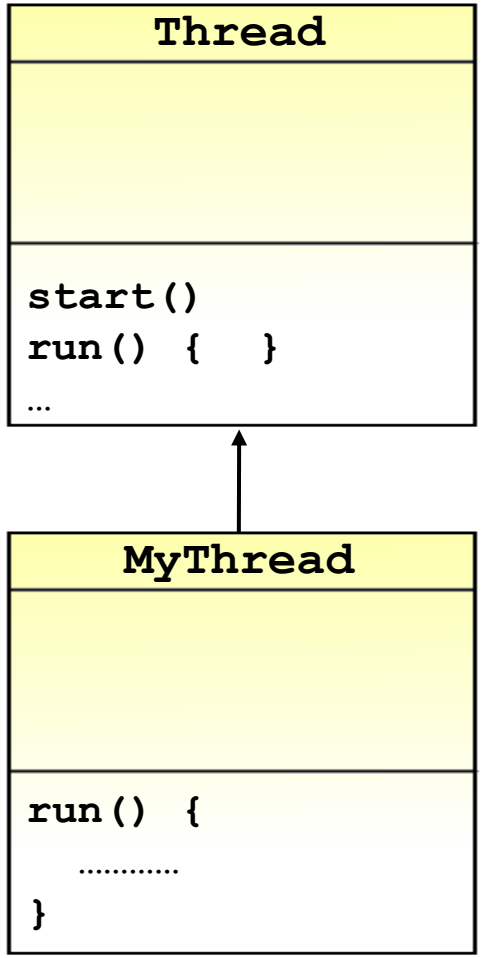
- Threads that are ready for execution are put in the ready queue.
  - Only one thread is executing at a time, while the others are waiting for their turn.
- The task that the thread carries out is written inside the `run()` method.

The `Thread` Class

- **`Class Thread`**
  - `start()`
  - `run()`
  - `sleep()*`
  - `suspend()*`
  - `resume()*`
  - `stop()*`

Working with Threads

- There are two ways to work with threads:
  - Extending Class **`Thread`**:
    1. Define a class that extends **`Thread`**.
    2. Override its **`run()`** method.
    3. In main or any other method:
       a. Create an object of the subclass.
       b. Call method **`start()`**.



Working with Threads

```
public class MyThread extends Thread
{
   public void run()
   {
    … //write the job here
   }
}
```

1

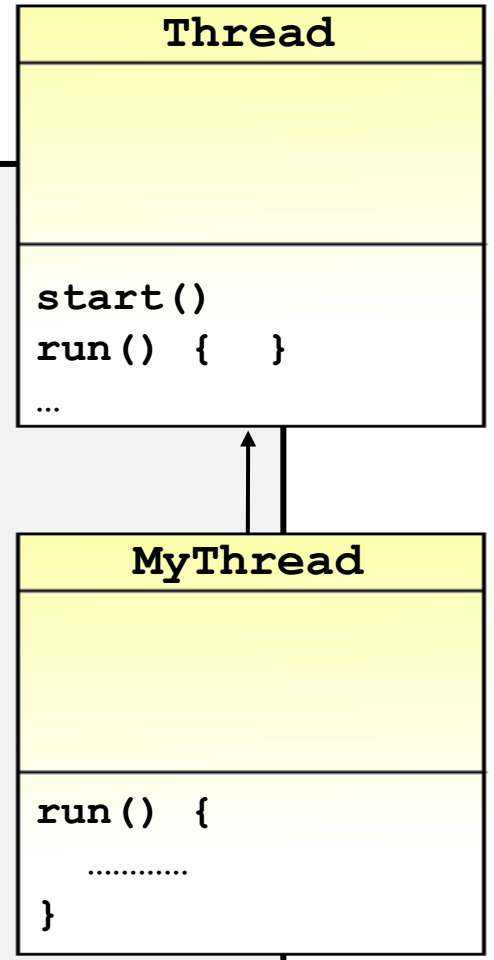
2

- in **main()** or any method:

```
public void anyMethod()
{
   MyThread th = new MyThread();
   th.start();
}
```

3. a

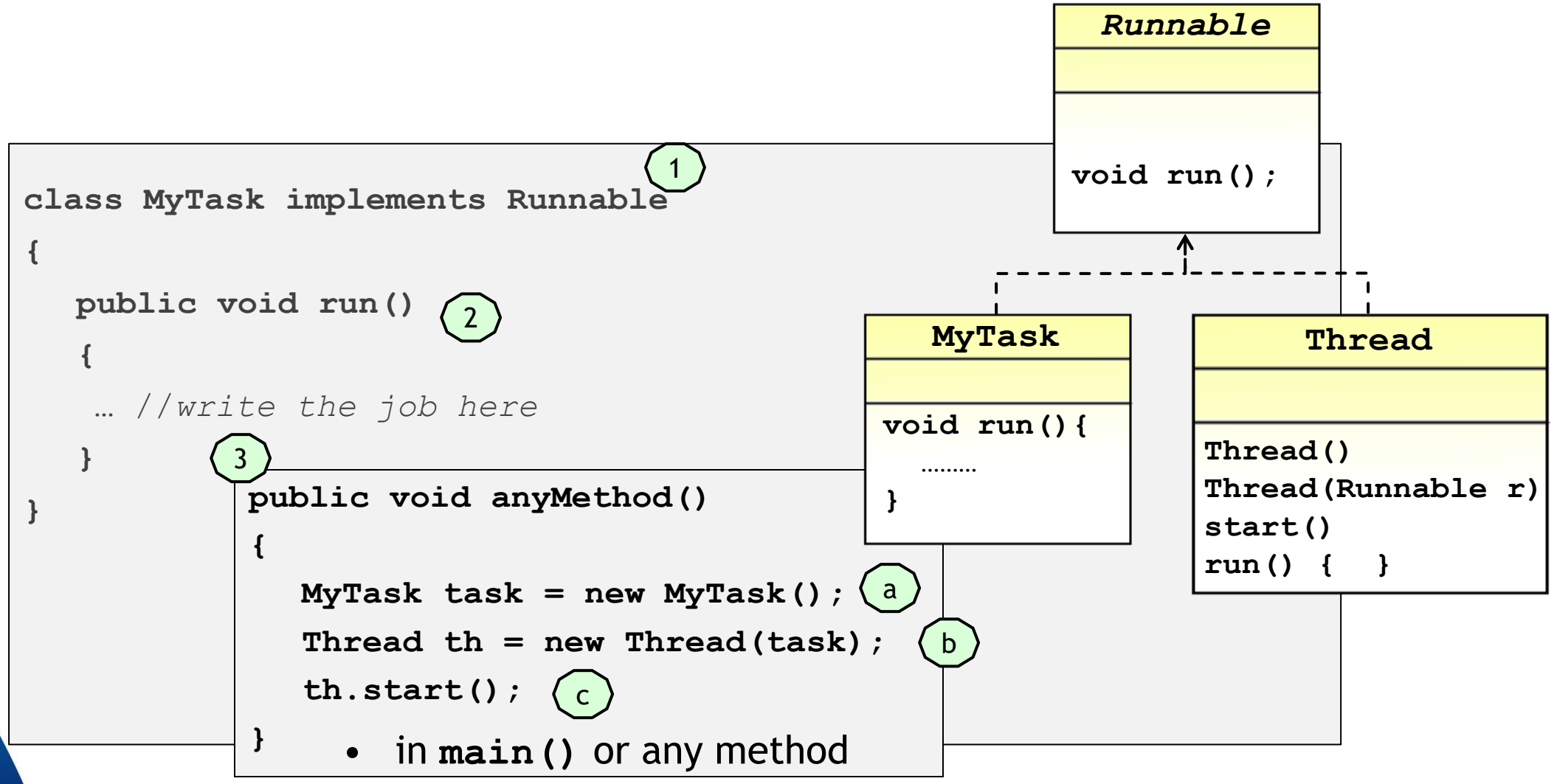
3. b



Working with Threads

- There are two ways to work with threads:
  - Implementing Interface ***`Runnable`***:
    1. Define a class that implements **`Runnable`**.
    2. Override its **`run()`** method .
    3. In main or any other method:
       a. Create an object of your class.
       b. Create an object of class **`Thread`** by passing your object to the constructor that requires a parameter of type **`Runnable`**.
       c. Call method **`start()`** on the **`Thread`** object.

Working with Threads



Extending Thread VS. Implementing Runnable

- Choosing between these two is a matter of taste.
- Implementing the Runnable interface:
  - May take more work since we still:
    - Declare a Thread object
    - Call the Thread methods on this object
  - Your class can still extend other class
- Extending the Thread class
  - Easier to implement
  - Your class can no longer extend any other class

Example:

```
public class SimpleThread extends Thread {
    String name;
    public SimpleThread(String name) {...3 lines }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + name);
            try {
                sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("DONE! " + name);
    }

}
```

```
0 Thread Object 1
1 Thread Object 1
2 Thread Object 1
3 Thread Object 1
4 Thread Object 1
5 Thread Object 1
6 Thread Object 1
7 Thread Object 1
8 Thread Object 1
9 Thread Object 1
DONE! Thread Object 1
```

```
public class Lecture_Demo {

    public static void main(String[] args) {

        new SimpleThread("Thread Object 1").start();
```

Example:

```
public class SimpleThread extends Thread {
    String name;
    public SimpleThread(String name) {...3 lines }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + name);
            try {
                sleep((int) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("DONE! " + name);
    }

}
```

```
0 Thread Object 1
0 Thread object 2
1 Thread object 2
2 Thread object 2
1 Thread Object 1
3 Thread object 2
4 Thread object 2
2 Thread Object 1
3 Thread Object 1
5 Thread object 2
6 Thread object 2
4 Thread Object 1
7 Thread object 2
5 Thread Object 1
8 Thread object 2
6 Thread Object 1
9 Thread object 2
DONE! Thread object 2
7 Thread Object 1
8 Thread Object 1
9 Thread Object 1
DONE! Thread Object 1
```

```
public class Lecture_Demo {

    public static void main(String[] args) {

        new SimpleThread("Thread Object 1").start();
        new SimpleThread("Thread object 2").start();
```

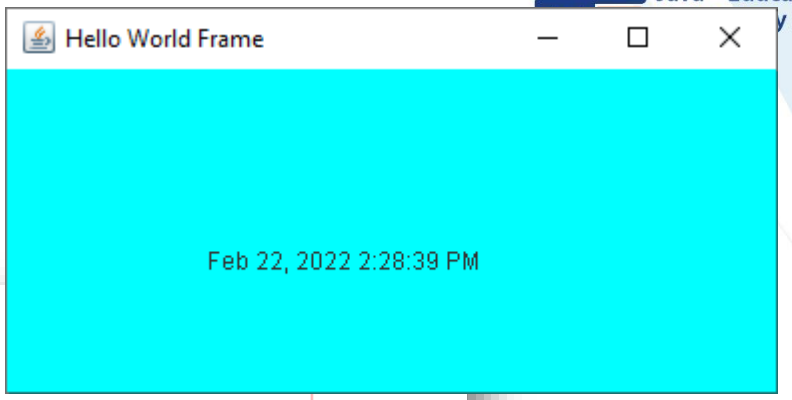
Example:

```
public class SimpleRunThread implements Runnable{
 String name;
    public SimpleRunThread(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + name);
            try {
                sleep((int) (Math.ra
            } catch (InterruptedExce
                e.printStackTrace();
            }
        }
        System.out.println("DONE! "
```

```
5 Thread object 2
6 Thread object 2
4 Thread Object 1
4 runnable object
7 Thread object 2
5 Thread Object 1
5 runnable object
6 Thread Object 1
7 Thread Object 1
6 runnable object
8 Thread object 2
8 Thread Object 1
9 Thread Object 1
9 Thread object 2
```

```
public static void main(String[] args) {

    new SimpleThread("Thread Object 1").start();
    new SimpleThread("Thread object 2").start();

    new Thread(new SimpleRunThread("runnable object")).start();

}
```

Create Simple GUI – java Swing Panel with Thread



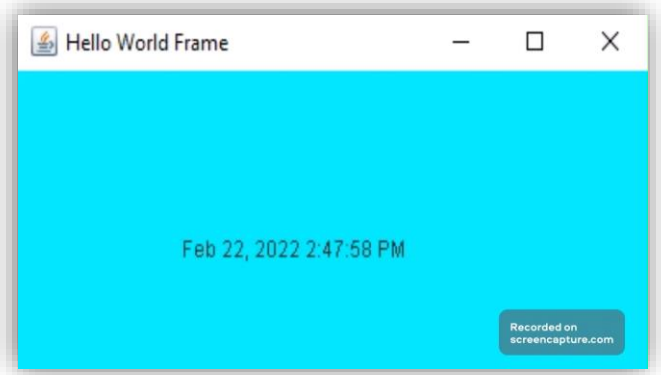

```
class MyPanel extends JPanel implements Runnable{

    public MyPanel() {
        this.setBackground(Color.cyan);
        new Thread(this).start();
    }
    @Override
    public void paintComponent(Graphics g) {...5 lines }
    @Override
    public void run() {
    while(true){
        try {
            this.repaint();
            Thread.sleep(1000);
        } catch (InterruptedException ex) {
            Logger.getLogger(MyPanel.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
    }
```

Java™ Education and Technology Services

Lab Exercise

Lab Exercise 1

- Simple Date and Time JFrame Application.

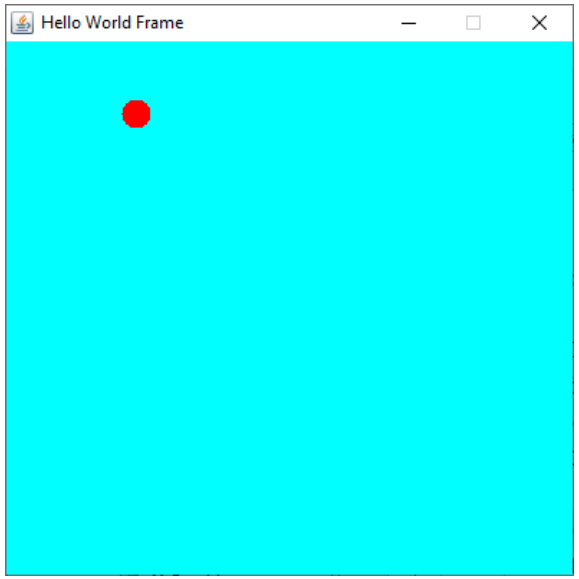


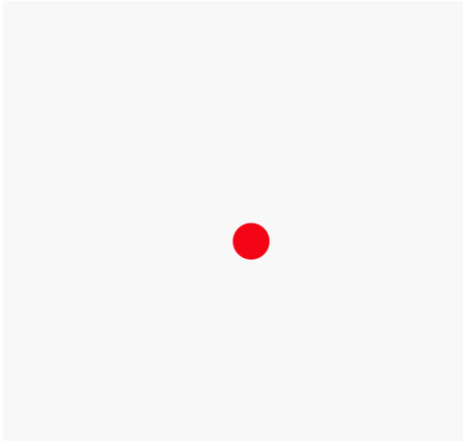
Lab Exercise 2

- Make a text **marquee** (is a scrolling piece of **text** displayed horizontally across your App.

Lab Exercise 3

- Make a BouncingBall App.





Lab Exercise 4

- Animate your lamp: